KFKI-76-28

L. VARGA

THE VDL GRAPH

Hungarian Academy of Sciences
CENTRAL
RESEARCH
INSTITUTE FOR
PHYSICS
BUDAPEST

KFKI-76-28

THE VDL GRAPH

L. Varga
Central Research Institute for Physics, Budapest, Hungary
M. T. C. Division
H-1525 Budapest, Box 49

ISBN 963 371 130 4

ABSTRACT

The VDL graph as an abstraction of data graphs is defined. A general graph walking algorithm is given, from which different concrete graph walking algorithms can be deduced. The linearly linked graph /chain/ and the binary tree as special cases of the VDL graph are defined. The use of the introduced objects in the definition of algorithms is illustrated by examples.

АННОТАЦИЯ

Определяем VDL граф и алгоритм обхода VDL графа. Определяем два специальных, важных с точки зрения практики, случая структуры данных. Линейную цепь и VDL дерево. Некоторые важные свойства определенных нами объектов объединяем в теоремах. Использование введенных объектов для решения задач определений иллустрируем примерами.

KIVONAT

Az adat-gráfok absztrakciójaként definiáljuk a VDL gráfot. Megadunk egy olyan általános gráf-bejárási algoritmust, amelyből különböző konkrét gráf-bejárási algoritmusok származtathatók. A lineárisan láncolt adatszerkezetet /a láncot/ és a lineáris fát mint a VDL gráf speciális esetét definiáljuk. A bevezetett objektumok algoritmusok definiálásában való felhasználását példákkal szemléltetjük.

1. Introduction

In the last decade a lot of work has been done in the field of the machine independent definitions of programming languages. One of the most remarkable results is the Vienna Definition Language /VDL/. An excellent treatment of VDL can be found in [1], [2], [3].

In the recent years the researchers tend to pay more and more attention to the abstract definitions of data structures. Such an abstraction describes the properties of the data structure independently from its concrete representation. Originally, VDL was introduced for the formal definition of tree-like data objects, as these type of objects were very well suited for the description of the formal syntax and semantics of programming languages. However, general data structures can be represented by graphs, rather than by trees. Therefore some authors approached the problem of the formal definition of abstract data structures in ways different from VDL. See for example [4], [5], [6]. In the latter work a rigorous definition of the formal semantics of data structures is given using the concept of the Common Base Language introduced by J.B.Dennis.

For the definition of data structures we need some suitably chosen basic objects from which any data structure can be built up. In [9] on the basis of practical considerations we introduced the VDL graph, and defined the abstract notion of data structure in the following way:

a data structure is an ordered triplet (t, S, K), where

$\text{is-structure}(t) = \text{TRUE}$,

S is the set of the selection operations over t,

K is the set of the construction operations over t,

and

$\text{is-structure} = \text{is-data-set} \vee \text{is-data-list} \vee \text{is-data-graph}$

$\text{is-data} = \text{is-structure} \vee \text{is-element}$

Here

$$\text{is-data-set} = (\{\langle s{:}\text{is-data}\rangle \mid \text{is-selector}(s)\})$$

and is-data-list is defined as usual in VDL.

In this paper the most important properties of the VDL graph are summarized. Two special cases of the VDL graph: the linearly linked data structure /chain/, and the VDL tree are defined. The use of the introduced objects in the solution of definitional problems is illustrated by examples.

2. The definition of the VDL graph

Let

$$\text{is-node-set} = (\{\langle s{:}\text{is-node}\rangle \mid \text{is-selector}(s)\})$$

$$\text{is-node} = (\langle \text{s-value:is-data} \vee \text{is-'NIL'}\rangle, \langle \text{s-desc:is-selector-list}\rangle)$$

where the null object is represented by the name NIL, and "is-selector" and "is-data" may represent arbitrary predicates.

Let

$$\text{is-node-set}(g) = \text{TRUE}$$

Definition 2.1. Let $t \in g$, if

$$(\exists s, \text{is-selector}(s) = \text{TRUE})(s(g) = t \text{ and } t \neq \text{NIL}).$$

Definition 2.2. Let $t \in g$, $n \in g$. The node n refers to t if and only if

$$(\exists i, 1 \leq i \leq \text{length}(\text{s-desc}(n)))((\text{elem}(i)(\text{s-desc}(n)))(g) = t).$$

Notationally we shall use the form

$$n \dashrightarrow t$$

Definition 2.3. We say that the node t_k is reachable from node t_1, or there exists a reference path from t_1 to t_k if and only if

$$t_1 \dashrightarrow t_2 \dashrightarrow \ldots \dashrightarrow t_k, \qquad /t_i \in g, \quad i=1,2,\ldots,k/$$

We shall use the following notation for the reference path:

$$t_1 \dashrightarrow^{*} t_k$$

Definition 2.4. The VDL graph is defined as an object which conforms to the following predicate

$$\text{is-data-graph} = (\{< \text{s-is-node} > \mid \text{is-selector}(s)\})$$
$$\text{is-node} = (< \text{s-value} : \text{is-data} >, < \text{s-desc} : \text{is-selector-list} >)$$

where if

$$\text{is-data-graph}(g) = \text{TRUE},$$

then there exists a subset of nodes

$$M = \{n \mid \text{is-master}(n)\} \subset \{n \mid \text{is-node}(n)\}$$

distinguished with the property that any node $n \in g$, / is-master(n)=FALSE/ can be reached from at least one element of m.

Definition 2.5. Let

$$\text{value}(n) = \text{s-value}(n)$$

and

$$\text{next}(i,n,g) = (\text{elem}(i)(\text{s-desc}(n)))(g)$$

if

$$\text{is-data-graph}(g) = \text{TRUE}$$

and

$$n \in g$$

Definition 2.6. A terminal node (n) is a node such that

$$\text{s-desc}(g) = < >$$

where

$$\text{is-data-graph}(g) = \text{TRUE}.$$

The structure of a data object can be visualised by a graph. The graph

$g = (< \text{s-r}_1 : n_1 >, < \text{s-r}_2 : n_2 >, < s_3 : n_3 >, < s_4 : n_4 >, < s_5 : n_5 >)$

$n_1 = (< \text{s-value} : a >, < \text{s-desc} : < s_3, s_4 >>)$

$n_2 = (< \text{s-value} : b >, < \text{s-desc} : < s_4 >>)$

$n_3 = (< \text{s-value} : c >, < \text{s-desc} : < >>)$

$n_4 = (< \text{s-value} : d >, < \text{s-desc} : < s_5, s_2 >>)$

$n_5 = (< \text{s-value} : e >, < \text{s-desc} : < >>)$

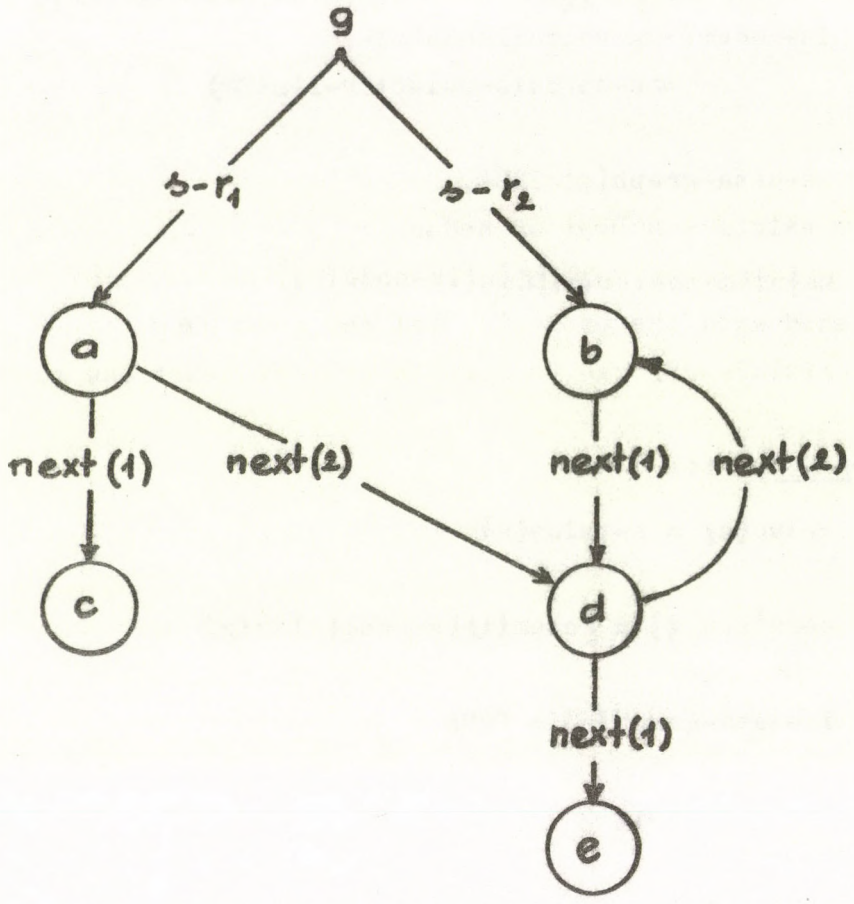


Fig. 2. 1.

is shown in Figure 2.1, where the directly reachable nodes of the VDL graph g can be selected by the selectors $s\text{-}r_1$, $s\text{-}r_2$. The nodes are circles in the Figure 2.1 and they contain a value. The values which are yielded by the nodes of the data graph can be selected by the selector s-value. For example

$$s\text{-}value(s\text{-}r_1(g)) = a \ ,$$

or simply

$$value(s\text{-}r_1(g)) = a \ .$$

The set, from which the values of the nodes are taken is not relevant here; it can be the set of the integer or real numbers, character strings etc.

The relationships between the nodes are represented by arrows. The arrows are named by function next(i). The selector function next(i) is defined as:

$$next(i)(n) = next(i,n,g).$$

Also composite selectors containing the selector function next(i) can be used, for example

$$value.next(1).next(2).\ s\text{-}r_1(g) = e$$

3. The graph walking

A fundamental operation is the graph walk. Most of the selection and construction operations can be established on the graph walk. A graph walk can be carried out according to different strategies. In the following we give an abstract graph walk algorithm from which the concrete graph walks can be deduced.

Let the state ξ of the abstract machine be defined as an object which conforms to the predicate

is-state = (<s-input:is-data-graph>,
<s-table:is-table>,
<s-control:is-control>)

where

$$\text{is-table} = (\{\langle s : \text{is-value} \rangle \mid \text{is-selector}(s)\})$$
$$\text{is-value} = \{T, F\}$$

Let the initial state ξ_0 of the machine be characterized by

$$\xi_0 = \mu_0(\langle \text{s-input} : g \rangle, \langle \text{s-table} : t_0 \rangle, \langle \text{s-control} : \underline{\text{walk}}\ (\text{s-input}\ (\xi_0)) \rangle)$$

where

$$\text{is-data-graph}(g) = \text{TRUE},$$
$$t_0 = (\{\langle s : F \rangle \mid \text{is-master}(s(g))\}).$$

Definition 3.1. The next-selector function is a function over the set

$$\{t \mid \text{is-table}(t)\}$$

The range of the function is

$$\text{is-selector} \vee \{\text{NIL}\}$$

If

$$(\exists s, \in \text{is-selector})(s(t) = F)$$

then

$$\text{next-selector}(t)(t) = F$$

else

$$\text{next-selector}(t) = \text{NIL}$$

Informally, the function next-selector(t) furnishes one of the selectors of table t for which

$$s(t) = F$$

if such an s exists, and yields the object NIL otherwise.

Lemma 3.1. Let

$$\text{is-data-list}(t) = \text{TRUE}$$

then the algorithm

```
process-list(t) =
 length(t) = 0 →null
 T →process-list (tail(t));
        process(head(t))
```

executes the instruction process exactly once for each element of list t.

Proof. This results from the definition of the functions "head" and "tail".

Theorem 3.1. Let ξ_0 be the object given above. The following program executes the instruction

process-value (n)

exactly once for each $n \in g$.

```
walk (g) =
next-selector(s-table (ξ)) = NIL →null
T →walk (g);
      process-value(n),
      process-selectors(w);
          w : next-selectors (n);
                n : next-node (next-selector(s-table (ξ)))

next-node (s) =
   PASS: s(s-input (ξ))
   s-table: μ(s-table (ξ) ; <s:T>)

next-selectors (n) =
   PASS: s-desc(n)

process-selectors (w) =
   length (w) = 0 →null
   T →process-selectors (tail(w));
        set (head(w))

set (s) =
   s(s-table (ξ)) = NIL →link (s)
   T →null

link (s) =
   s-table: μ(s-table (ξ) : <s:F>)
```

Proof. Let us prove that the control tree is reduced to the instruction null if and only if the instruction

process-value (n)

has been executed exactly once for each n∈g.

Let us suppose, that the control tree has been reduced to the instruction null. Then by Definition 3.1. there exists no selector s such that

s(s-table(ξ)) = F

This means, that instruction

next-node (s)

has been executed for each s, where

is-master (s(g)) = TRUE

But in this case the instructions

next-selectors(s(g))
process-value (s(g))

and

process-selectors (s-desc(s(g)))

also have been executed for each s, where

is-master (s(g)) = TRUE

The execution of the last instruction for each selector s, by which the node s(g) refers to a node n∈g, implies that

- if s(t)≠NIL then the table t does not change,
- if s(t)=NIL then the component <s:F> will be added to the table t.

Thus it is obvious that if the control tree has been reduced to the instruction null, then each node n∈g, for which

$m \longrightarrow^{*} n$, is-master(m) = TRUE

holds, has been processed. However it results from Definition 2.4., that then each node n∈g has been processed and the instruction

process-value (n)

has been executed exactly once for each n∈g.

Now let us supose that the instruction

process-value (s(g))

has been executed exactly once for each s(g)≠NIL. Let us prove that then the control tree has been reduced to the instruction null.

Each instruction

process-value(s(g))

is preceded by the instruction

next-node (s),

which results in changing the table t such a way that

s(t) = T.

Since the table s-table(ξ_0) contains all the "master" selectors, and during the execution of the control tree only such selectors can be entered into the table for which s(g)≠NIL, and further, these selectors are entered before processing the object s(g), it follows that after processing any s(g)

s(s-table(ξ)) = T

and hence the control tree must have been reduced to the instruction null. This completes the proof of Theorem 3.1.

The graph walking strategy is specified by the function

next-selector(t)

Hence the given algorithm with different mappings of the function next-selector(t) defines different graph walking strategies.

The graph walking problem frequently occurs in system's programming. Such a problem is for instance linking relocatable binary segments together to form a complete program. In the following the algorithm of an abstract linkage editor program will be given.

5.1 An abstract linkage editor

Let us see a system in which the segments can refer to each other only by segment names.

Definition 3.2.

is-r/b-program = is-segmentcode-graph

where

is-selector = is-segmentname

In detail:

is-r/b-program = ({ <s:is-node> | is-segmentname(s) }),
is-node = (< s-value : is-segmentcode>,
< s-desc : is-segmentname-list >)

Definition 3.3. Let

editor(t)

be the macro instruction which processes a segmentcode as needed for linking. The concrete structure of it is irrelevant here.

Definition 3.4. Let

is-r/b-program(p) = TRUE.

The abstraction of a linkage editor is the following graph walking algorithm:

```
linker(p) =
next-selector(s-table(§)) = NIL →null
T →link (p);
        editor (n),
        process-selectors (w);
           w: next-selectors (n);
                 n: next-node (next-selector(s-table(§)))
```

The macro instructions

process-selectors
next-selectors
next-node

are defined, as in Theorem 3.1. The details of an abstract linkage editor can be found in [8].

4. Linearly linked data structure /chain/

The graph has two special cases, which are of particular importance in defining data structures. These are the chain and the tree.

Definition 4.1. The chain is defined as an object conforming to the predicate

is-data-chain.

is-data-chain(t) = TRUE

if and only if

1. is-data-graph(t) = TRUE,
2. There exists exactly one $m \in t$ such that
   is-master (m) = TRUE,
   /Let "first" be the selector, for which
   is-master(first(t)) = TRUE./
3. There exists exactly one $n \in t$, which is terminal,
4. If $n \in t$ and n is not a terminal, then
   length (s-desc(n)) = 1.

Theorem 4.1. If

is-data-chain (t) = TRUE,

$n \in t$, $n \neq$ first (t),

then there exists exactly one reference path from first(t) to n.

Proof. Definition 2.4. states that there exists at least one reference path from first(t) to n:

first (t) $\rightarrow^*$ n

Thus we have to prove that the reference path is unique. This results from the fact, that if $n \in t$ and n is nonterminal, then

length (s-desc(n)) = 1

and futhermore, there exists a $v \in t$ which is terminal.

Definition 4.2. Let

is-data-chain(t) = TRUE.

Let us define the following functions:

1. If $n \in t$, then

   next(n,t) = next(1,n,t)

2. If $s(t) \in t$ and n=next(s(t),t), then

   cancel(n,t)=

   μ(t;<s-desc.s: s-desc(n)>,<elem(1)(s-desc(s(t))) :NIL >)

3. If is-data(a) = TRUE and $s(t) \in t$ then

   put-next(a,s(t),t)=

   μ(t;<s' :μ_0 (<s-value:a>,<s-desc:s-desc(s(t))>)>,

   <s-desc.s:<s'>>),

   where s'(t) = NIL.

4. If $n \in t$, then

   cut(n,t) = ({<s:s(t)>| s(t)-->*n})

   From these definitions we may deduce the following lemmas:

Lemma 4.1.

is-data-chain(cancel(n,t))

Lemma 4.2.

is-data-chain(put-next(a,n,t))

Lemma 4.3.

is-data-chain(cut(n,t))

The algorithm of chain walking is very simple:

walk-chain(t) = walk-node(first(t),t)

walk-node(n,t) =

n = NIL → null

T → walk-node (next(n,t),t);

process-value(n)

The operation of searching for the terminal element of a chain is significant in practice. This problem can be solved by the chain walking algorithm. However it is preferable to define a function which maps a chain to its terminal element.

Definition 4.3. If

is-data-chain (t) = TRUE,

then

$$\text{last}(t) \in t$$

and

$$\text{next}(\text{last}(t)) = \text{NIL}$$

The introduction of this function facilitates appending a new element at the end of a chain or deleting the last element:

put-next(a, last(t), t),
cut(last(t), t).

5. The binary tree

Another important special case of the graph is the binary tree. An abstraction of the data tree can be found in [3]. In the following we define the binary tree as a special case of the VDL graph.

Definition 5.1. A binary tree may be represented by the object conforming to the predicate:

is-data-tree.
is-data-tree(t) = TRUE

if and only if

1. is-data-graph(t) = TRUE,
2. There exists exactly one $n \in t$ such that
   is-master(t) = TRUE
   /Let "root" be the selector, for which
   is-master(root(t)) = TRUE /
3. If $n \in t$, then
   length(s-desc(n)) ≤ 2
4. If $n \in t$, $n \neq$ root(t), then these exists exactly one reference path from root(t) to n.

Definition 5.2. Let

is-data-tree(t) = TRUE

Let us define the following functions:

1. If $n \in t$, then
   left(n,t) = next(1,n,t)

and

$$right(n,t) = next(2,n,t)$$

2. If $is\text{-}data(a) = TRUE$, $s(t) \in t$ and $left(s(t),t) = NIL$

then

$$put\text{-}left(a,s(t),t) = \mu(t;\langle s' : \mu_0(\langle s\text{-}value:a\rangle,\langle s\text{-}desc:\langle\rangle\rangle)\rangle, \langle elem(1)\bullet s\text{-}desc\bullet s:s'\rangle),$$

where $s'(t) = NIL$

3. If $is\text{-}data(a) = TRUE$, $s(t) = t'$ and $right(s(t),t) = NIL$

then

$$put\text{-}right(a,s(t),t) = \mu(t;\langle s' : \mu_0(\langle s\text{-}value:a\rangle,\langle s\text{-}desc:\langle\rangle\rangle)\rangle, \langle elem(2)\bullet s\text{-}desc\bullet s:s'\rangle),$$

where $s'(t) = NIL$

Definition 5.3. If

$$is\text{-}data\text{-}tree(t) = TRUE$$

then

$$left\text{-}tree(t) = (\{\langle root:n_1\rangle\} \cup \{\langle s:s(t)\rangle \mid n_1 \rightarrow^* s(t)\})$$

where

$$n_1 = (elem(1)(s\text{-}desc(root(t))))(t)$$

and

$$right\text{-}tree(t) = (\{\langle root:n_2\rangle\} \cup \{\langle s:s(t)\rangle \mid n_2 \rightarrow^* s(t)\})$$

where

$$n_2 = (elem(2)(s\text{-}desc(root(t))))(t)$$

Theorem 5.1. Let $is\text{-}data\text{-}tree(t) = TRUE$.

a/ If $left(root(t)) \neq NIL$, then
$is\text{-}data\text{-}tree(left\text{-}tree(t)) = TRUE$

b/ If $right(root(t)) \neq NIL$, then
$is\text{-}data\text{-}tree(right\text{-}tree(t)) = TRUE$

c/ If $n \in t$, $n \neq root(t)$, then
or $n \in left\text{-}tree(t)$,
or $n \in right\text{-}tree(t)$.

Proof. It follows from Definition 2.4 and Definition 5.1/3 that for each $n \in t$, $n \neq root(t)$, $n \neq n_1$, $n \neq n_2$

or $root(t) \rightarrow n_1 \rightarrow^* n$,

or $root(t) \rightarrow n_2 \rightarrow^* n$.

Hence part c/ of Theorem 5.1. is true.

Now let us see the object

left-tree(t).

It is obvious that for this object the statements 2. and 3. in Definition 5.1. hold. On the other hand, it follows from Definition 5.3 that

$(\forall n, n \in left\text{-}tree(t))(\ n_1 \rightarrow^* n\ )$.

Hence

is-data-graph(t) = TRUE

Since only one reference path exists from root(t) to n and this reference path is the following

$root(t) \rightarrow n_1 \rightarrow^* n$

it can be seen, that

$n_1 \rightarrow^* n$

is unique. Similarly the correctness of statement c/ is also obvious.

Theorem 5.2. If is-data-tree(t) = TRUE, $n \in t$, $n \neq root(t)$ and n is not a terminal, then there exists at least one reference path such that

$root(t) \rightarrow^* n \rightarrow^* v$

where v is terminal.

Proof. Based on the Definition 5.1. we may see that

$root(t) \rightarrow^* n$

If n is not terminal then at least one of the objects left(n,t) and right(n,t) is not NIL. Let n' be such an object. Then

$root(t) \rightarrow^* n \rightarrow n'$

If n' is not a terminal, since there is no reference path such that

$\bar{n} \rightarrow^* \bar{n}$

repeating this procedure, we have

$root(t) \rightarrow^* n \rightarrow n' \rightarrow^* n_k$

where n_k is a terminal.

There are three wellknown tree walking algorithms which are important in practice: the postorder, preorder and endorder algorithm /see [7]/. With the operations defined above as a basis, we can describe these algorithms very simply. Let us see the postorder algorithm.

Definition 5.3. Let

is-data-tree(t) = TRUE

The postorder tree walking algorithm can be formulated as follows:

```
walk-tree-post(t) =
  t = NIL →null
  T →walk-tree-post(right-tree(t));
        process-value(root(t));
          walk-tree-post(left-tree(t))
```

where operation process-value can be replaced with an arbitrary operation.

Theorem 5.3. If is-data-tree(t) = TRUE, then the program

walk-tree-post(t)

executes the instruction

process-value(n)

once and only once for each $n \in t$.

Proof. Since

$root(t) \notin$ right-tree(t),
$root(t) \notin$ left-tree(t)

the instruction

process-value(root(t))

is executed by the program exactly once. Similarly it can be seen that the instructions

process-value (root(left -tree(t))),
process-value (root(right-tree(t))),

and process-value((n))

for each

$$n = root(*(...*(t)))$$

where $*$ is replaced by the functions left-tree and right-tree, are executed exactly once. But obviously

$$n = root(*(...*(t))) \neq NIL$$

if and only if

$$root(t) \rightarrow\rightarrow * \; n \; .$$

This completes the proof.

5.1 Sort tree

We often need a structure the elements of which can be accessed in less time than those of an unordered list and which permits the insertion of new elements easily. A sort tree is one such structure /see [10]/. In the following we give an algorithm for creating a sort tree as a part of a sorting algorithm.

Let t be given, where

is-data-list(t) = TRUE.

Let us assume that t is an unordered list. Let us define a transformation which transforms the list t to an ordered list t'. A list t' is ordered, if and only if

$$(\forall i, 1 \leq i < length(t'))(elem(i)(t') \leq elem(i+1)(t'))$$

The state of the abstract machine (ξ) may be specified as an object which conforms to the following predicate

is-state = (<s-input:is-data-list>,
<s-output:is-data-list>,
<s-tree:is-data-tree>,
<s-control:is-control>)

and the initial state is as follows

ξ_0 = (<s-input:t>,
<s-output:<>>
<s-tree:μ_0(<root:μ_0(<s-value:head(t)>)>)>,
<s-control:sort (tail(t))>),

where

```
is-data-list(t) = TRUE
```

The abstract sorting algorithm:

```
sort(t) =
length(t) ≠ 0 →sort (tail(t));
                    set-tree (head(t),root(s-tree(ξ)))
T →walk-tree-post (s-tree (ξ))
```

where now the instruction process(v) is replaced with

```
process(v) =
  s-output: s-output (ξ)⌢<v>
```

```
set-tree (v,n) =
  value(n)< v →test-right (v,n)
  value(n)≥ v →test-left (v,n)
```

```
test-right (v,n) =
    right(n,s-tree (ξ)) ≠ NIL →set-tree (v,right(n,s-tree(ξ)))
    T → s-tree: put-right(v,n,s-tree (ξ))
```

```
test-left (v,n) =
    left(n,s-tree (ξ)) ≠ NIL →set-tree (v,left(n,s-tree(ξ)))
    T →s-tree: put-left(v,n,s-tree (ξ))
```

The proof of the correctness of this algorithm is theoretically simple but a little bit laborious and therefore it is omitted here.

References

[1] Neuhold, E.J.: The formal description of programming languages.
IBM System Journal 10. /1971/

[2] Wegner, P.: The Vienna definition language.
Computing Surveys 4. /1972/

[3] Lee, A.N.: Computer semantics.
Van-Nostrand Reinhold Co. /1972/

[4] Early, J.: Toward an understanding of data structures.
Communications of ACM 14. /1971/

[5] Hoare, C.R.A.: Notes on data structuring,
Structured programming. Academic Press /1972/

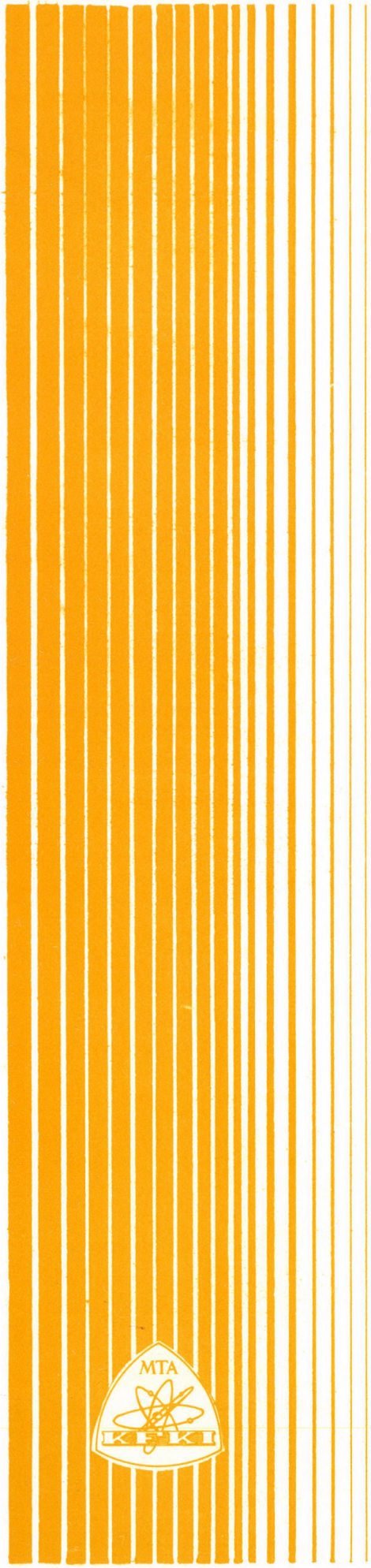
[6] Ellis, D.J.: Semantics of data structures and references.
MC-16375 /1974/

[7] Knuth, D.: The art of computer programming.
Addison-Wesley, Vol.1. /1969/

[8] Varga, L.: The abstractions of machine dependent program forms.
KFKI-76-11 /1976/

[9] Varga, L.: Abstract syntax and semantics of data structures /in Hungarian/
Alkalmazott Matematikai Lapok /to be published/

[10] Berztiss, A.T.: Data structures, theory and practice.
Academic Press /1971/



Kiadja a Központi Fizikai Kutató Intézet
Felelős kiadó: Sándory Mihály igazgató
Szakmai lektor: Lőcs Gyula
Nyelvi lektor : Lőcs Gyula
Példányszám: 405 Törzsszám: 76-403
Készült a KFKI sokszorosító üzemében
1976. május hó